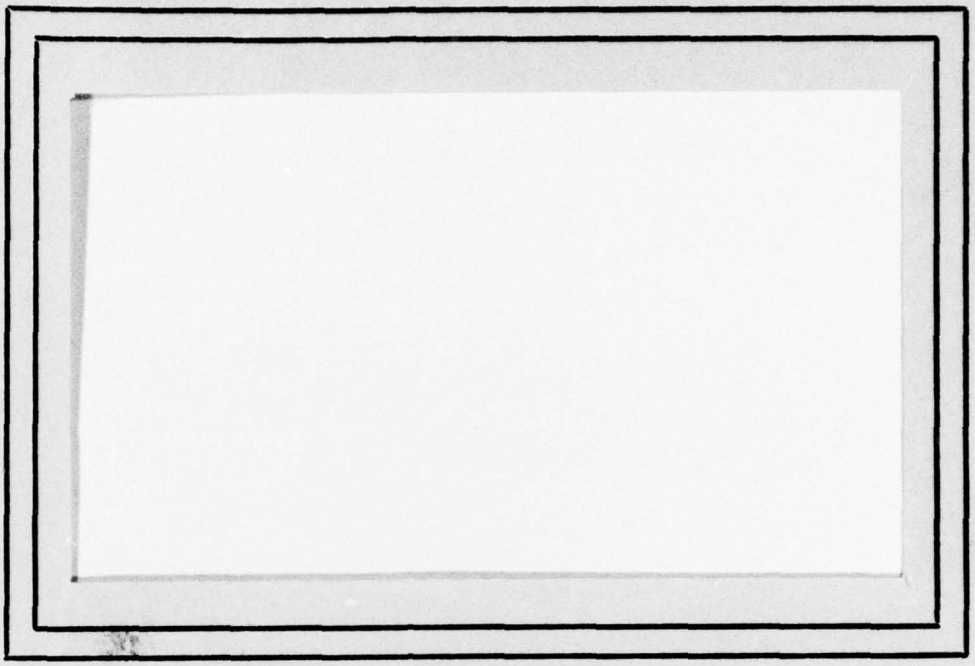

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(1)

(14) TR-612
DAAG53-76C-0138,
(15) ARPA Order-3206

(11) Dec[redacted]77

(12) 69 p.

(6) PDP 11 Image Processing Software.

(10) Kenneth C./Hayes, Jr.,
Martin/Herman
Russell/Smith

Computer Vision Laboratory
Computer Science Center
University of Maryland
College Park, Md. 20742

(9) Technical rept.,

ABSTRACT: The implementation of two different image handling systems for use on PDP 11/45 minicomputers is documented. Micro-XAP is an image access facility for UNIX FORTRAN users. Virtual Arrays are a more general image access facility for UNIX LISP users. This report serves as a user's guide for users who will interact with Micro-XAP or Virtual Arrays.

The support of the U. S. Army Night Vision Laboratory under Contract DAAG53-76C-0138 (ARPA Order 3206) is gratefully acknowledged, as is the help of Azriel Rosenfeld, Phil Dondes, and Sunny Banvard.

403 018

Image Processing Software under UNIX

Table of Contents

1. Introduction

A number of image processing packages run on the Computer Vision Laboratory's PDP 11/45. Micro-XAP and Virtual Arrays are basic image processing packages that run under the UNIX operating system. Mini-XAP is a robust image processing system that runs under the DOS operating system. Since the Laboratory only runs UNIX, the Mini-XAP system is not currently operated and will not be documented here.

Micro-XAP, a collection of FORTRAN callable image access routines, will be presented first. Then the Virtual Array package (VAP), a collection of LISP callable image access software, will be described. Some general software support packages for LISP users will be documented in the final section.

2. Micro-XAP

Micro-XAP is a small collection of image access and creation routines similar to the kernal routines of 1108 XAP [1]. The routines were originaly coded by Marty Herman and have since been modified by Russ Smith. The routines are FORTRAN callable under the UNIX operating system. Output pictures are created by calling the routines:

a) SETUPW once,

b) PWRITE once for each output row,

c) XCLOSE once.

Input pictures are accessed by calling the routines:

a) SETUPR once,

b) IPREAD once for each input row,

c) XCLOSE once.

There are two utility routines, HEADER, and SCANOUT, for acquiring picture header information and displaying pictures on the CRT respectively.

There are three string manipulation routines, PNAME, NAMCAT, and LENGTH.

The last routines described are VPRINT, VP, PHIST, and 1PER16, all commands to the UNIX shell written by Russ Smith. VPRINT and VP display pictures on the system's PRINTRONIX line printer. PHIST lists histograms of pictures on the printer while 1PER16

takes upto 20 input pictures and produces a display of their composition on the CRT display.

2.0.1. SETUPW

CALL: I = SETUPW (AREA,NAME,IW,SHIFT,BYPX)

VALUE: 1 if call is successful, -1 if an error occured

EFFECT: The picture NAME is associated with AREA and opened for sequential writing (by PWRITE). Flags are set to indicate that the picture NAME is open for writing not for reading. SETUPW builds a XAP header for the output picture. The window IW and the bytes per pixel are saved in the header. The window IW is checked to make sure all its values are greater than or equal to zero. Next, it is determined if the output file for the picture NAME exists. If it does not, it is created. The window IW, the file description for the picture file NAME, and the SHIFT are saved in AREA. Finally, the header is written out to the picture file.

NOTES: AREA is a 15 word INTEGER*4 work area for XAP

NAME is a 16 word INTEGER*4 path name to a UNIX file that will be created or which already exists. NAME is large enough to hold a 63 character path name.

IW is a 4 word INTEGER*4 defining the dimensions of the output picture. IW(3) is the number of columns in the picture. Iw(4) is the number of rows in the picture.

SHIFT is an INTEGER*4. Each picture point value written to picture NAME is multiplied by 2**SHIFT.

BYPX is an integer*4. BYPX is the number of bytes per picture point in the output picture. BYPX may be 1,2, or 4.

2.U.2. PWRITE

CALL: I = PWRITE (AREA,NUM,VECTOR)

VALUE: 1 if call is successful, -1 if an error occured

EFFECT: PWRITE writes out the next row of a picture. If NUM is less than 1, then PWRITE returns. The column dimension (IW(3)) of the output picture is taken from AREA. PWRITE outputs picture point values VECTOR(1), ... ,VECTOR(IW(3))*NUM) as the next NUM rows of the picture associated with AREA before returning. Before being written out, each point in VECTOR is multiplied by 2**SHIFT, where SHIFT is a parameter stored in AREA by SETUPW. Each group of IW(3) words in VECTOR is compressed according to the value of BYPX that was also stored in AREA by SETUPW.

NOTES: SETUPW must be called before PWRITE is called.

AREA is a 15 word INTEGER*4 work area for XAP.

NUM is an INTEGER*4. NUM is the number of rows in VECTOR to be written out to the picture associated with AREA.

VECTOR is an INTEGER*4 vector. VECTOR should be at least IW(3)*NUM elements. VECTOR contains the picture point values to be written out to the picture associated with AREA.

2.0.3. XCLOSE

CALL: I = XCLOSE (AREA)

VALUE: 1 if call is successful, -1 if an error occured

EFFECT: The picture file associated with AREA is closed. If the user failed to complete the window IW associated with AREA, then XCLOSE prints a warning message. If the window was not completed and AREA is associated with an output file, then execution is terminated. XCLOSE returns.

NOTES: SETUPR or SETUPW must be called befor XCLOSE.

AREA is a 15 word INTEGER*4 work area for XAP.

2.0.4. SETUPR

CALL: I = SETUPR (AREA,NAME,IW,SHIFT,DEPTH,PTR)

VALUE: 1 if call is successful, -1 if an error occured

EFFECT: The picture NAME is associated with AREA and opened for sequential reading (by IPREAD). Flags are set to indicate that the picture NAME is open for reading but not for writing. The XAP header is read in and the following parameters are saved in AREA: The number of columns, number of rows, bytes per pixel, the file descriptor for the picture NAME, the window, and SHIFT. The user input window IW is checked against the window from the header. If the window IW is invalid or DEPTH is less than 1, then an error message is printed and execution is terminated. The window and bytes per pixel obtained from the header are used to initialize control

parameters stored in AREA. These parameters allow IPREAD to extract the appropriate columns (as specified by IW(1) and IW(3)) from an input row for storage in the array passed to IPREAD. The vector PTR is initialized by setting PTR(I) to I, where I = 1,...,DEPTH.

NOTES: AREA is a 15 word INTEGER*4 work area for XAP.

NAME is a 16 word INTEGER*4 path name to an existing XAP picture. This path name can be up to 63 characters long.

IW is a 4 word INTEGER*4 vector contain a window specification for NAME.

SHIFT is an INTEGER*4. Each picture value is multiplied by 2**SHIFT before it is placed in ARRAY.

DEPTH is an INTEGER*4. DEPTH is the maximum number of rows of picture NAME user will keep in array ARRAY.

PTR is a DEPTH word INTEGER*4 vector of subscript values. PTR(1) is the subscript of the least recently read row. PTR (DEPTH) is the subscript of the most recently read row.

2.0.5. IPREAD

CALL: I = IPREAD (AREA,NUM,DEPTH,PTR,ARRAY,RLENGTH)

VALUE: 1 when row(s) properly read in, 0 when no more rows in window, -1 when an error.

EFFECT: If NUM is less than 1, then PREAD returns a zero immediately. If all of the picture rows in the window IW associated with AREA have been passed to the user,

then IPREAD returns the value 1, otherwise it returns 0. If at least one row of the window of the input picture associated with AREA remains to be read, then the row is input. The control parameters stored in AREA (and calculated by SETUPR) are used to unpack the points in the input row belonging to the window IW. The points in the window are multiplied by 2**SHIFT as they are copied to ARRAY (1,PTR(1)),...,ARRAY(IW(3),PTR(1)). The PTR vector associated with AREA is then rotated. That is, PTR(I) is set to PTR(I+1), for I = 1,...,DEPTH-1. PTR(DEPTH) is set to the original value of PTR(1). The dimension of the array ARRAY associated with AREA is RLENGTH x DEPTH. The above procedure is performed NUM time before IPREAD returns unless the window is completed before NUM rows are processed. It should be noted that at most DEPTH rows of the input picture are in ARRAY at any one time. The oldest row (nearest the top of the window) begins at ARRAY (1,PTR(1)) and the newest row (lowest row read in the window so far) begins at ARRAY (1,PTR(DEPTH)).

NOTES: SETUPR must be called before IPREAD is called.

AREA is a 15 word INTEGER*4 work area for XAP.

NUM is an INTEGER*4 containing the number of rows of the picture associated with AREA to be read into the array ARRAY.

DEPTH is an INTEGER*4 containing the maximum number of rows of picture NAME the user will keep in array ARRAY.

PTR is a DEPTH word INTEGER*4 vector of subscript values. PTR(1) is the subscript of the least recently read row. PTR(DEPTH) is the subscript of the most recently read row.

ARRAY is an RLENGTH x DEPTH word INTEGER*4 array to hold rows of picture NAME.

RLENGTH is an INTEGER*4 that is the row length of ARRAY and the maximum window width SETUPR and IPREAD will allow.

2.0.6. HEADER

CALL: I = HEADER (INAME,HBUF)

VALUE: 1, if there is no I/O error while reading the header record of file INAME, otherwise header terminates.

EFFECT: The header information on (picture) INAME is stored in the vector HBUF. If file INAME does not exist or an I/O error occurs while HEADER reads the header record, then an error message is printed and execution is terminated.

NOTES: INAME is a 16 word INTEGER*4 vector containing a path name to a UNIX file.

HBUF is a 3 word INTEGER*4 vector.

HBUF(1) is the column length of the picture.

HBUF(2) is the row length of the picture.

HBUF(3) is the number of bytes per pixel for the picture (only 1,2, or 4).

2.0.7. SCANOUT

CALL: SCANOUT

VALUE: None

EFFECT: SCANOUT prompts the user for a picture file name and a window specification. SCANOUT copies the window of the picture to the photographic and video display. The scanner must be properly initialized.

NOTES: The picture values should be in the range 0 to 63.

2.0.8. PNAME

CALL: CALL PNAME (TPLATE,NAME,NUM)

VALUE: None

EFFECT: The characters in TPLATE are concatenated with the ASCII character representation of the number NUM and stored in NAME. NAME can be used as a XAP picture name.

NOTES: TPLATE is a UNIX path name. TPLATE is assumed to be dimensioned large enough to contain 48 characters.

NAME is assumed to be dimensioned large enough to contain 64 characters.

NUM is a single word integer.

The completed XAP picture name must be less than 64 characters long.

2.0.9. NAMCAT

CALL: CALL NAMCAT (NAME1,LENGTH1,NAME2,LENGTH2,CATNAME)

VALUE: None

EFFECT: The character string in NAME1 of length LENGTH1 is

concatenated with the string NAME2 of length LENGTH2 and stored in the vector CATNAME. A blank character is added to the string in CATNAME so that the string in CATNAME can be used as a UNIX file name.

NOTES: NAME1 is an integer vector containing a character string. NAME1 is assumed to be dimensioned large enough to hold a string of length LENGTH1.

NAME2 is an integer vector containing a character string. NAME2 is assumed to be dimensioned large enough to hold a string of length LENGTH2.

CATNAME is an integer vector to contain the concatenation of NAME1 and NAME2. CATNAME is assumed to be dimensioned large enough to hold a string of length LENGTH1 + LENGTH2 + 1.

EXAMPLE: The following code saves the concatenation of the two string "/dir" and "/pct" in INAME.

```
INTEGER N1(16),N2(16),INAME(16)
DATA N1(1)/"/dir"/
DATA N2(1)/"/pct"/
CALL NAMCAT (N1,4,N2,4,INAME) which is equivalence to
                                    doing
CALL NAMCAT ("/dir" 4 "/pct" 4 INAME)
```

2.0.10. LENGTH

CALL: V = LENGTH (INAME,NLENGTH)

VALUE: The length of the string in INAME.

EFFECT: LENGTH returns the (byte) length of the character string

stored in the integer vector INAME. NLENGTH is assumed to be the dimension (INTEGER*4) of INAME.

NOTES: INAME and NLENGTH are both INTEGER*4 input arguments. They are not modified.

2.0.11. VPRINT

CALL: % vprint [N or -N] picture-name1 ... picture-name<n>

EFFECT: VPRINT lists out the picture files picture-name1 ... picture-name<n>. The pixels of each picture are shifted N places to the left (or right when a dash "-" precedes the number N) before the four least significant bits of each shifted pixel are used to select one of sixteen possible grey levels for display on the PRINTRONIX line printer. All pictures are assumed to be one byte per pixel. Each picture file must be in PDP/XAP format. Pictures that are wider than 64 pixels will be printed in 64 column strips. The complete image can be viewed by assembling the picture strips together.

NOTES: If no shift is given, then the first picture name cannot begin with a dash "-" or a digit.

2.0.12. VP

CALL: % vp [N or -N] picture-name1 ... picture-name<n>

EFFECT: VP lists out the picture files picture-name1 ... picture-name<n>. The pixels of each picture are shifted N places to the left (or right when a dash "-" precedes the number N) before the four least significant bits of

each shifted pixel are used to select one of sixteen possible grey levels for display on the PRINTRONIX line printer. All pictures are assumed to be one byte per pixel. Each picture file must be in PDP/XAP format. Pictures that are wider than 128 pixels will be printed in 128 column strips. The complete image can be viewed by assembling the picture strips together.

NOTES: If no shift is given, then the first picture name cannot begin with a dash "-" or a digit.

2.0.13. PHIST

CALL: % phist [-] picture-name1 ... picture-name<n>

EFFECT: A histogram for each PDP/XAP formatted picture picture-name1 ... picture-name<n> is printed. If the dash "-" is present, then a listing of the histograms is sent to the standard output. If the dash "-" is missing, then a listing of the histograms including bar graphs will be plotted on the PRINTRONIX line printer.

NOTES: If the optional dash "-" is missing, then the first picture name cannot begin with a dash.

2.0.14. 1PER16

CALL: % 1per16

EFFECT: 1per16 is an interactive routine to create one CRT display image from up to twenty different PDP/XAP formatted input pictures. 1per16 will place the pictures in the display image or the user can manually

specify the composition of the display. Any row of the display can have pieces of at most eleven different input pictures.

NOTES: The CRT display should be turned on, set online to the PDP11/45, and set up to accept the desired picture size.

3. Virtual Arrays

The Virtual Array package (VAP) is a collection of image access and creation routines. The routines are LISP functions running under the UNIX operating system. The elements of a virtual array (pixels) can be N bytes long, where N is a power of two. When elements are accessed (row or points), the Virtual Array package returns values as either ARRAYs or VECTORs. So elements can be accessed as integer, real, double precision, or complex values, or as vectors of binary, integer, real, double precision, or complex values. Unfortunately, LISP SYS calls only work with ARRAYs; thus writing values out to a virtual array is more complicated than it need be.

The following functions are defined by the Virtual Array package.

3.0.1. VAC: Virtual Array Creation

CALL: (VAC 'VAID FILE-NAME X-SIZE Y-SIZE BYTE-SIZE
[POINT-ROW? [MIN-X MAX-X MIN-Y MAX-Y]])

VALUE: The pointer array that is constantly bound to VAID

EFFECT: VAC creates the file FILE-NAME and allocates enough space in FILE-NAME to maintain an X-SIZE by Y-SIZE virtual array. When BYTE-SIZE is positive, each element of the virtual array is 2**(BYTE-SIZE) bytes long. When BYTE-SIZE is negative, the elements are (length 1) ARRAYs or VECTORs, which associate a data type with the

data in the virtual array. The initial value of each element is zero. After creating the file, VAC closes the file (VAX) then re-opens it using VAO. See VAO's description.

NOTES: FILE-NAME is a string node defining a valid UNIX file.

X-SIZE is the maximum allowable index for the x dimension (legal subscripts run from 0 through X-SIZE).

Y-SIZE is the maximum allowable index for the y dimension (legal subscripts run from 0 through Y-SIZE).

BYTE-SIZE allows the user to define the form of the elements of the virtual array. If BYTE-SIZE is positive, then each element is 2**(BYTE-SIZE) bytes long and VAP routines return data (VAID 6) in a byte ARRAY. If BYTE-SIZE is greater than -11, then each element is a (length 1) VECTOR with type |BYTE-SIZE| and VAP routines return data in an appropriate VECTOR. If BYTE-SIZE is less than -10, then each element is a (length 1) ARRAY with type |BYTE-SIZE - 20| and VAP routines return data in an appropriate ARRAY.

The purpose of allowing negative valued BYTE-SIZE arguments is that it associates a type with the data in a virtual array and automatically initializes VAID at open time, so that users can reference the data properly. For example, a virtual array's elements would be word integer (array), double-precision integer (vector), or complex (vector), if BYTE-SIZE had value -25, -6, or -9, respectively.

POINT-ROW? is an optional argument. If this argument is missing or NIL, then the virtual array is opened for point accessing (i.e. all read and write requests access a single element of the virtual array). If POINT-ROW? is non-NIL, then the virtual array is opened for row accessing and all read and write functions (except VA and VAW) transfer a row of element from the window of the virtual array.

MIN-X, MAX-X, MIN-Y, MAX-Y define a window of the virtual array. "read(write)-next" and "read(write)-row" functions won't reference points outside of the window. "read(write)-point" functions will reference any point fo the virtual array regardless of the window. The default values for missing window specification arguments are 0, X-SIZE, 0, Y-SIZE, respectively.

3.0.2. VAO: Virtual Array Opening

CALL: (VAO 'VAID FILE-NAME [POINT-ROW?
[MIN-X MAX-X MIN-Y MAX-Y]])

VALUE: The pointer array that is constantly bound to VAID

EFFECT: VAO closes any virtual array currently associated with VAID. Then VAO opens the virtual array stored in the file with name FILE-NAME and associates it with VAID. If the file does not exist, then VAO prints an error message and then evaluates an (error 20). VAO reads in a 6-byte header record that contains the values X-SIZE, Y-SIZE, and BYTE-SIZE; these were set at creation time.

The window is defaulted to the entire virtual array. If POINT-ROW? is absent, then VAO conditions VAID for point accessing and returns. Otherwise, the value of POINT-ROW? is checked. If it is NIL, then VAID is conditioned for point accessing, else VAID is conditioned for row accessing. Now, VAO modifies the window by as many arguments (MIN-X ... MAX-Y) as the user supplies. If this new window extends beyond the boundary of the virtual array, then VAO evaluates an (ERROR 20). Otherwise, VAO returns to the user.

NOTES: FILE-NAME is a string node defining a valid UNIX file.

X-SIZE is the maximum allowable index for the x dimension (legal subscripts run from 0 through X-SIZE).

Y-SIZE is the maximum allowable index for the y dimension (legal subscripts run from 0 through Y-SIZE).

For discussion of BYTE-SIZE see NOTES of VAC.

MIN-X, MAX-X, MIN-Y, MAX-Y define a window of the virtual array. "read(write)-next" and "read(write)-row" functions won't reference points outside of the window. "read(write)-point" functions will reference any point to the virtual array regardless of the window.

3.0.3. VA: Virtual Array Reading (Point)

CALL: (VA VAID X Y)

VALUE: The array (or vector) containing the value of the point at coordinates (X,Y) of the virtual array associated with VAID.

EFFECT: The element (X,Y) of the virtual array associated with VAID is loaded into the array (VAID 5). VA returns the ARRAY or VECTOR in (VAID 6) to the user. The two arrays (VAID 5) and (VAID 6) are assumed to reference the same storage area.

NOTES: X is an index for the x dimension. X is not checked for being in range.

Y is an index for the y dimension. Y is not checked for being in range.

EXAMPLE: The following code will make a histogram of picture "/pict" sampled every other column and every other row. The picture is assumed to be one byte per picture point.

```
(VAO 'ID "/pict")
(setq HIST (vector 255 6))
(mapn 0 (ID 2) (lambda (-y)
     <cond [<zerop (logand 1 -y)>]
           [T <mapn 0 (ID 1) (lambda (-x)
                     <cond [< zerop (logand 1 -y)>]
                           [T
                              <setq I (<VAID -x -y>1)>
                              <HIST I (add1 <HIST I>)>]>
                     )>]>
     ))
(VAX 'ID)
```

3.0.4. VAW: Virtual Array Writing (Point)

CALL: (VAW VAID X Y VALUE)

VALUE: The array VALUE

EFFECT: VAW replaces element (X,Y) of the virtual array associated with VAID by VALUE. VALUE is assumed to be an ARRAY large enought to hold at least one element of the virtual array. VAW returns VALUE.

NOTES: X is an index for the x dimension. X is not checked for being in range.

Y is an index for the y dimension. Y is not checked for being in range.

EXAMPLE: The following code will place a 5x5 square with value 64 in the picture "/pict". The square will be centered at point (10,15) of "/pict". "/pict" is assumed to be a byte picture.

```
(VAO 'ID "/pict")
(setq POINT (Array 1 4))
(POINT 1 64)
(mapn 13 17 (lambda (-y)
      <mapn 8 12 (lambda (-x)
            <VAW ID -x -y POINT>
            )>
      )>
(VAX 'ID)
```

3.0.5. VAR: Virtual Array Reading (Row)

CALL: (VAR VAID Y)

VALUE: The array (or vector) containing the Yth row of the virtual array associated with VAID.

EFFECT: VAR reads row Y of the window defined on the virtual array associated with VAID into the array in (VAID 5). If VAID was opened for point accessing, then VAR reads in the point at (MIN-X,Y). If row Y is outside the user defined window, then VAR evaluates an (ERROR 20). VAR returns (VAID 6).

NOTES: (VAID 5) is a byte array for a point (or row of points). (VAID 6) is an array (or vector) of user chosen type that is equivalenced to (vaid 5).

EXAMPLE: The following code opens the virtual array "/pictures/s1", reads in and lists its first row, and then closes the virtual array.

```
(VAO ID "/pictures/s1" t)
(VAR ID 1)
(cond[<arrayp (vaid 6)> <cseta TOP (arrayl (vaid 6))>
                        <cseta BOT 1>]
     [t <cseta TOP (VECTORl (vaid 6))>
        <cseta BPT 0>])
(cseta R nil)
(mapn TOP BOT (lambda (-i)
      <seta R (cons <(vaid 6) -i> R)>))
(print R)
(VAX 'ID)
```

3.0.6. VAWR: Virtual Array Writing (Row)

CALL: (VAWR VAID Y VALUE)

VALUE: The array VALUE

EFFECT: VAWR replaces row Y of the window defined for the virtual array associated with VAID by the elements in VALUE. If VAID was opened for point accessing, then VAWR replaces the point at (MIN-X,Y). If row Y is outside of the user defined window, the VAWR evaluates an (ERROR 20). VAWR returns VALUE.

NOTES: VALUE is an ARRAY (it cannot be a VECTOR) containing a row of elements. The data in VALUE are written out to the virtual array.

EXAMPLE: The following code will produce a 15x15 picture with a 5x5 square (with value 64) centered in it. Each element of the virtual array is a byte long. The background has value zero.

```
(VAC 'ID "/square",14 14 0 T 5 9 5 9)
(cseta ROW (Array 5 4))
(mapn 1 5 (lambda (-X)
      <ROW -x 64 >))
(mapn 5 9 (lambda (-y)
      <VAWR ID -y ROW>))
(VAX 'ID)
```

3.0.7. VAN: Virtual Array Reading (Next Row or Point)

CALL: (VAN VAID)

VALUE: The array (or vector) containing the next point (VAID opened for point accessing) or the next row of points (VAID opened for row accessing) of the virtual array

associated with VAID.

EFFECT: VAN reads the next point (VAID opened for point accessing) or the next row (VAID opened for row accessing) into the ARRAY in (VAID 5). If the point or row is outside of the window defined on the virtual array, then VAN evaluates an (ERROR 20). VAN returns (VAID 6), an array (or vector) containing the next point or row.

EXAMPLE: The following code computes the average value of a picture ("/pict") whose elements are one byte long.

```
(VAO 'ID "/pict")
(csetq SUM (csetq NPT (Double 0)))
(attempt [prog <>
          TOP <csetq SUM (plus SUM <(VAN ID) 1 >)>
              <csetq NPT (add1 NPT)>
              <go TOP>]
         [20])
(VAX 'ID)
(print (cons "AVERAGE VALUE IS" (quotient SUM NPT)))
```

3.U.8. VAWN: Virtual Array Writing (Next Row or Point)

CALL: (VAWN VAID VALUE)

EFFECT: VAID replaces the next point (VAID opened for point accessing) or the next row (VAID opened for row processing) of the user window of the virtual array associated with VAID with the element(s) stored in the array VALUE. If the point or row is outsidee of the

user defined window, then VAWN evaluates an (ERROR 20). VAWN returns VALUE.

NOTES: VALUE is an ARRAY (it can not be a VECTOR) containing a single element (if VAID opened for point accessed) or a row of elements (if VAID was opened for row accesses). The data in VALUE are written out to the virtual array.

EXAMPLE: The following code will place a 5x5 square with value 64 in the picture "/pict". The square will be centered at point (10, 15). "/pict" is assumed to be a byte picture.

```
(VAO 'ID "/pict" nil 8 12 13 17)
(csetq POINT (Array 1 4))
(POINT 1 64)
(attempt [prog < >
         TOP <VAWN ID POINT>
             <go TOP>]
        [20])
(VAX 'ID)
```

3.0.9. NVA: Virtual Array Reading (Next Point)

CALL: (NVA VAID)

EFFECT: NVA reads in the next element of the virtual array associated with VAID. NVA does no window checking and does not keep track of the coordinates of the currently accessed point of the virtual array.

3.0.10. NVAW: Virtual Array Writing (Next Point)

CALL: (NVAW VAID [optional arguments])

EFFECT: NVAw writes out the data stored in (VAID 5) as the value of the next element of the virtual array associated with VAID. NVAW does no window checking and does not keep track of the coordinates of the currently accessed point of the virtual array.

3.0.11. SVA: Updating a Point or Row Value

CALL: (SVA VAID [optional arguments])

VALUE: The array (or vector) containing the updated row or point of the virtual array associated with VAID

EFFECT: SVA replaces the last referenced elements (VAID opened for point accessing) or row (VAID opened for row accessing) of the virtual array associated with VAID with the current contents of the array (VAID 5). SVP returns (VAID 6).

NOTES: (VAID 5) is a byte array equivalenced with the array or vector in (VAID 6).

EXAMPLE: The following code will add one to each point in the byte picture "/pictures/p1".

```
(VAO 'ID "/pictures/p1")
(ATTEMPT [prog < >
          TOP <SVA ID (<ID 5> 1 <add1 (<VAN ID > 1)>)>
              <go TOP>]
         [20])
(VAX 'ID)
```

3.0.12. VAX: Virtual Array Closing

CALL: (VAX 'VAID)

VALUE: The pointer array that has been constantly bound to VAID.

EFFECT: VAX closes the file associated with the virtual array controlled by VAID, if there is one. VAX returns the value of VAID (a pointer array). VAX evaluates an (ERROR 20) if its input argument is not an atomic symbol. VAX initializes VAID, if neccessary, by making its value a 16 place pointer array of NILs.

NOTES: After the virtual array associated with VAID is closed, no input or output (ex. VA or VAW) should be performed with VAID until some virtual array is opened (VAO) and associated with VAID. Failure to obey this restriction will result in unusual behavior on the standard input file (file descriptor 0), especially when a program is run in background.

3.0.13. VAID: Virtual Array IDentifier

VAID stands for Virtual Array IDentifier. VAID's value is a 16 place pointer array whose values are as follows: (VAC VAO VAX initialize this variable, not user)

1 is maximum x dimension value (X-SIZE, or # columns - 1).

2 is maximum y dimension value (Y-SIZE, or # rows - 1).

3 is log base 2 of the byte length of a virtual array element (computed from BYTE-SIZE).

4 is the UNIX file descriptor for the virtual array.

5 is a byte ARRAY big enough to hold a single element or a row of elements, depending on whether VAID was opened for point access or row access.

6 is initialized to be identical to (VAID 5). If the user wants to receive an ARRAY or VECTOR structure other than a byte array, then he should modify (VAID 6). For example, <VAID 6 (RPLACA (ARRAY 1 5) (*CAR (VAID 5)))> would allow the user to directly reference elements of a virtual array if BYTE-SIZE was 1.

7 is the x coordinate of the beginning of the last point(s) accessed.

8 is the y coordinate of the last row accessed.

9 is the byte length of records to be read or written. For point accessing this is the length of a single element. For row accessing this is [MAX-X - MIN-X + 1] times the length of a single element.

10 is the byte length of a row of elements (of the virtual array, not the window).

11 is a position vector used for "seeking" around the virtual array file.

12 is the POINT-ROW? flag. If point accessing was requested, then (VAID 12) is NIL, otherwise, it is non-NIL.

13 is the minimum x coordinate a "read-next", "write-next", "read-row", or "write-row" function can access in the virtual array.

14 is the maximum x coordinate a "read(write)-next" or a "read(write)-row" function can access in the virtual array.

15 is the minimum y coordinate a "read(write)-next(row)" function can access in the virtual array.

16 is the maximum y coordinate a "read(write)-next(row)" function can access in the virtual array.

3.0.14. VPSK: Virtual Array Seek Utility

CALL: (VPSK VAID X Y)

VALUE: Non-NIL

EFFECT: VPSK conditions the virtual array file of VAID so that

the next read or write will effect the element at coordinates (X,Y). VPSK calculates the displacement of element (X,Y). If the file is already at that position, then VPSK returns. If the file is positioned near the element, then VPSK does a relative seek to the new position. Otherwise VPSK does an absolute seek to the new position. This last seek is actually two seeks, because the byte displacement can be greater than a PDP 11 word. VPSK remembers the position of the file as being the beginning of the addressed element.

NOTES: Users should never have to call this routine.

3.0.15. VADUMP: List Status of VAID

CALL: (VADUMP 'VAID)

VALUE: NIL

EFFECT: The contents of the virtual array identifiers VAID are listed out. See VAID for a complete description of the form of VAID.

3.0.16. PRTOUT

CALL: (PRTOUT FILE-NAME IOT [DIR [WINDOW]])

VALUE: NIL

EFFECT: PRTOUT takes the virtual array in FILE-NAME and displays it on the standard output file. The standard output file is assumed to be a 132 column ASCII printer. The virtual array is opened (VAO) with the WINDOW parameters (MIN-X MAX-X MIN-Y MAX-Y). If the virtual array window

is wider than 126 columns, then PRTOUT prints the array out in vertical strips, so that the whole window is reproduced. If DIR is absent or NIL, then the virtual array is printed out from lower row numbers to higher row numbers; if DIR is non-NIL, then the array is displayed in the opposite order. Each array element in the window is modified by the function (of one variable) IOT. If IOT is NIL, then function MANIFEST is used instead. PRTOUT assumes that IOT returns a LISP object that requires only a single print position when the object is passed to PRIN1.

NOTES: FILE-NAME is a string node defining a valid UNIX file. IOT must be cognizant of the byte-size and type (floating or integer, etc) of data in the virtual array. WINDOW is defined by the 4-tuple MIN-X MAX-X MIN-Y MAX-Y. The default values for missing window specification arguments are 0, X-picture-dimension, 0, y-picture-dimension, respectively.

EXAMPLE: The following code will display a "chain code" picture on the printer. Assume that the picture is named "/chain" and that each of its picture points has three fields. The first field (F) is 1 when the point is part of the foreground (curve or intersection) and 0 when the point is part of the background. The second field (I) is 1 when two or more curves have crossed the point and 0 when at most one curve has crossed the point. The third field (C) is the chain code value for those points

on a curve and 0 for backgound points. The C field is the least significant 3 bits of a point. The I field is the next most significant bit. The F field is the next most significant bit above the I field. The chain code direction and print symbols are defined as follows:

```
7 0 1      ` ^ ´        background points are "b"
6 * 2      <   >        intersection points are "#"
5 4 3      / v \
```

```
<csetq CIOT (lambda (-v)
      <cond [<zerop (logand 20q - v)> "b"]
            [<zerop (logand 10q - v))> "#"]
            [T <car (nth '(!^ !´ !> !\ !v !/ !< !`)
                        (add1 (logand 7q -v)))>]>
      )>
<PRTOUT "/chain" CIOT>
```

3.0.17. SCANOUT

CALLS: (SCANOUT FILE-NAME IOT [DIR [WINDOW]])
(SCANOUT FILE-NAME-LIST IOT [DIR [WINDOW]])

VALUE: NIL

EFFECT: SCANOUT takes the virtual array(s) in FILE-NAME(-LIST) and displays them on the scanner. The scanner is assumed to be set up to receive a 256x256 picture. All virtual arrays are opened (VAO) with the same WINDOW parameters (MIN-X MAX-X MIN-Y MAX-Y). If DIR is absent or NIL, then the array(s) are scanned out from lower row numbers to higher row numbers. If DIR is non-NIL, then

the array(s) are displayed in the opposite order. Each array element is modified by the function (of one variable) IOT. If IOT is NIL, then the function MANIFEST is used instead.

NOTES: FILE-NAME is a string node defining a valid UNIX file.

IOT must be cognizant of the byte-size and type of data in the virtual array.

WINDOW is defined by the 4-tuple MIN-X MAX-X MIN-Y MAX-Y. The default values for missing window specification arguments are 0, X-picture-dimension, 0, Y-picture-dimension, respectively.

EXAMPLE: The following code will display a picture (with less than 257 columns) on the scanner. The picture name is "/pict". Each element of "/pict" is one byte long. The grey values of "/pict" range from 0 to 15. The IOT in this example will extend the range of grey values from 0 to 255 (really 240).

```
(cseta IOT (lambda (-v) <times 16 -v>))
(scanout "/pict" IOT)
```

4. The VECTOR Package

4.1. Introduction

While PDP 11 LISP is a powerful programming language, our experience writing picture processing software revealed four areas for improvement in LISP. The improvements eliminate the inefficient or awkward language constructs that are necessary to perform some basic picture processing tasks. This section describes the LISP functions implemented in the VECTOR package to alleviate these deficiencies. The VECTOR package software is loaded by evaluating the expression (load "/lsp/vct") after typing % lisp to the UNIX SHELL. The following paragraphs will describe features of the VECTOR package.

4.1.1. Extended data types

Single (or multi) dimensioned arrays are not standard LISP data structures. However, many LISP systems implement arrays for user convenience. PDP 11 LISP supports pointer, logical (T or NIL), signed and unsigned byte, integer word, floating point, and double precision floating point single dimensional arrays. The picture processing software accommodates more data types, such as signed and unsigned binary data, double precision integer data, and single and double precision complex numbers. Simulating these data types with PDP 11 LISP arrays proved to be quite

awkward. Also the arithmetic functions that handled these simulated data types were complicated and inefficient. This data type problem was eliminated by programming a new function named VECTOR to create singly dimensioned arrays for all the data types used by Mini-XAP and Virtual Arrays. A set of arithmetic functions extending several basic arithmetic operations to these new data types were also constructed.

4.1.2. Iteration

Although it can be argued that iteration is inherently non-LISP-like or less elegant than recursion, picture processing routines commonly apply the same operation iteratively to sub-regions of pictures. Consequently, effective picture processing requires efficient methods of iteration. Iteration is often performed by manipulating indices, as in FORTRAN DO loops. The function MAPN, in file /lsp/mapn, will call a given function once for each integer in a range.

4.1.3. Node allocation

LISP's method of evaluation creates many data nodes that are used for a short period of time and then discarded. For example, a new data node must be allocated for every reference to an element of a LISP array (except pointer arrays, byte arrays, and word arrays when the value is pre-allocated). Every LISP function that returns a number as a result, also must allocate a new data node to hold that result. Whenever the result of a

calculation is returned to an element of an array and no intermediate results were CSETQed or SETQed, then all those node generations represent wasted effort. This waste manifests itself in two ways. One way is the time it took to allocate the node that will be discarded. The other way is by reducing the amount of available storage, and thus hastening the execution of a time consuming garbage collection process to retrieve all the discarded nodes.

The VECTOR package allays the node allocation problem by three separate measures. First, a few VECTOR package functions return NIL instead of a number as their result. The second measure the VECTOR package takes to reduce the node allocation problem is to define new argument protocols so that VECTOR package functions can get argument values directly without the overhead of an intermediate node generation to contain the argument value. For example, the LISP syntax for adding two elements of a vector together is (PLUS (V I) (V J))), where V is a vector and I and J are numbers. But the evaluation of (V I) and (V J) causes the value nodes to be generated before the function PLUS can be called. ADD is the VECTOR package "equivalent" to PLUS. The expression (ADD (V I) (V J)) would also generate two value nodes before function ADD is called. But the expression (ADD V I V J) will return the same value without the overhead of the two node generations. Note also that the second ADD expression requires two fewer implicit calls to EVAL.

The third measure used by the VECTOR package to reduce node

allocation is to allow the results of a VECTOR package function to be stored directly in an element of a vector. For example, the expression (V I (INC V I)) increments the value of the Ith element of vector V and returns the new value of the Ith element. The evaluation of (INC V I) causes the generation of a data node for the result of incrementing the value in the Ith element of V. This data node is returned as the value of the entire expression. The function VSET allows a VECTOR package function to be called without the generation of a data node for the result. The expression (VSET V I INC V I) increments the Ith element of V without an extra node generation. However, the returned value of the VSET expression is V. If a result is being stored in a vector and the result value is not needed, then the use of the function VSET will help reduce needless node allocation. It also executes faster than the nested expression.

4.1.4. Minor extentions

The set of LISP arithmetic functions and predicates is incomplete. For example, there is a less-than predicate (LESSP) but no less-than-or-equal predicate (defined by <lambda (X Y) <not (greaterp X Y)>>). Only the predicates for zero, minus, equal, greater-than, and less-than are defined in PDP 11 LISP. Several useful arithmetic functions are undefined, e.g., absolute value or minimum (of a set of numbers). The VECTOR package implements a complete set of predicates (EQL, NEQ, GT, GE, LT, LE, EQU, NEU, GTU, GEU, LTU, LEU) defined over both the standard LISP data types and the extended VECTOR package data types. The

following arithmetic functions are also implemented by the VECTOR package: ADD, SUB, MULT, DIV, MIN, MAX, INC, DEC, ABS, NEGATIVE. The following conversion functions are implemented: INT, REAL, IMAGINARY, COMPLEX. Some integer and logical operations are also implemented.

4.2. VECTOR Package Functions

4.2.1. Conventions

Some of the variables, appearing in the documentation of this appendix, have special meanings:

FCT — denotes a LISP function, system or user defined (interpretive Lambda, or compiled).

VFCT — denotes a function of the VECTOR package.

I — denotes an index used to reference an element of a vector. I is implicitly a number (node).

V — denotes a vector function, that is, the name of a single dimensional array created by the function VECTOR.

A,B or A1 ... — An denote arguments to VECTOR package functions. VECTOR package arguments are defined differently than ordinary LISP arguments. A VECTOR package argument is identical to a LISP argument if the argument is not a linker node. However, a VECTOR package argument can require two LISP arguments. This occurs when the first LISP argument is a vector, V (created by VECTOR) and the second LISP argument is a number node, I. VECTOR package functions treat this pair of arguments as a request for the value of the Ith element of the vector V.

4.2.2. Vector functions

4.2.2.1. VECTOR

CALL: (vector SIZE TYPE)

VALUE: A one dimensional vector (a special kind of function) of length SIZE+1 and of type TYPE.

EFFECT: Space for the vector is allocated by the same method the LISP function ARRAY uses.

NOTES: Vector elements are numbered from zero to SIZE. If SIZE is negative, the value zero is substituted for SIZE. The legal types (and the corresponding values of TYPE) of vectors are:

0	Signed bit (0 or -1)
1	Unsigned bit (0 or 1)
2	Signed byte (-128 to 127)
3	Unsigned byte (0 to 255)
4	Signed word
5	Signed word
6	Signed double word (integer)
7	Floating point
8	Double precision floating point
10	Double precision complex

If TYPE is not between 0 and 10, the value 5 is substituted for TYPE.

Vectors are functions; the rules for the evaluation of vectors are as follows. Assume that V and U are vectors (SETQed from (VECTOR SIZE TYPE)). Let I and J be numbers

such that -1 < I < (ADD1 (VECTORL V)) and -1 < J < (ADD1 (VECTORL U)). Let # represent a number and S represent a string. Then:

(V) evaluates to V and (U) evaluates to U.

(V I) evaluates to the contents of the Ith element of V. This value is returned as an integer node #, if its value is small enough, or as a string (node) S containing the type of the value as well as the value itself.

(V I #) evaluates to # and replaces the Ith elemtnent of V with V's representation for the value #.

(V I S) evaluates to S and replaces the Ith element of V with V's representation for the value in S.
(V I (U J)) is a special case of (V I #) or (V I S) because (U J) evaluates to a number # or a string S.
(V I U J) evaluates to NIL. The Ith element of V is replaced by the value of the Jth element of U.

when an element of a vector is being set to some value, that value is converted to the representation that corresponds to the type of the vector. Thus, if V is a floating point vector, then (V 1 0) would store a floating point zero in the first element of V.

4.2.2.2. VECTORP

CALL: (vectorp V)

VALUE: The type, an integer between 0 and 10, of the vector V or NIL if V is not a vector.

4.2.2.3. VECTORL

CALL: (vectorl V)

VALUE: The number of the highest element of vector V or NIL if

V is not a vector

NOTES: (VECTORL V) returns 1 fewer than the number of elements in V because vectors have a zero element.

4.2.2.4. CLEARV

CALL: (clearv V)

VALUE: NIL

EFFECT: The vector V is cleared to zero.

NOTES: V must be a VECTOR, not an ARRAY.

4.2.2.5. VSET

CALL: (vset V I VFCT A1 ... An)

VALUE: The vector V.

EFFECT: The result of applying the VECTOR package function VFCT to the arguments A1 ... AN is stored in the Ith element of vector V.

NOTES: The above call is equivalent to (OR (V I (VFCT A1 ... An)) V) but VSET generates no needless data node for the result of VFCT.

V must be a vector
I must be integer valued
VFCT must be one of the following functions:
ADD, SUB, MULT, DIV, MOD, MIN, MAX
IADD, ISUB, LAND, LEQV, LOR, LXOR,
ABS, DEC, INC, NEGATIVE,
INT, REAL, IMAGINARY, COMPLEX.

4.2.3. Iteration function

4.2.3.1. MAPN

CALL: (mapn BEGIN END FCT)

VALUE: NIL

EFFECT: The function of one variable FCT is called first with

value BEGIN, then once for each integer value between BEGIN and END, and finally with value END. IF BEGIN and END are the same number, then FCT is called just once.

NOTES: If FCT is not a function, then LISP will request the user to supply a function.

BEGIN and END should be integer valued; MAPN does not check.

If BEGIN is less then END, then MAPN increments the index value passed to FCT, otherwise MAPN decrements it.

Before a function that uses MAPN is compiled, the file "/lsp/fix" should be loaded. This allows the LISP compiler to compile MAPN calls of internal lambda functions without declaring the free variables to be fluid.

MAPN can be loaded from the file "/lsp/mapn".

4.2.4. Predicates

4.2.4.1. GT

CALL: (gt A B)

VALUE: T, if argument A's numeric value is greater than argument B's.

NIL, otherwise.

NOTES: Similar to LISP predicate GREATERP

4.2.4.2. GE

CALL: (ge A B)

VALUE: T, if argument A's numeric value is greater than or equal to argument B's.

NIL, otherwise.

NOTES: Similar to (OR (GREATERP A B) (EQUAL A B))

4.2.4.3. EQL

CALL: (eql A B)

VALUE: T, if argument A has the same numeric value as argument B.

NIL, otherwise.

NOTES: Similar to LISP predicate EQUAL applied to numeric arguments.

4.2.4.4. NE

CALL: (ne A B)

VALUE: T, if argument A's numeric value is not equal to argument B's.

NIL, otherwise.

NOTES: Equivalent to (NOT (EQL A B))

4.2.4.5. LE

CALL: (le A B)

VALUE: T,if argument A's numeric value is less than or equal to argument B's.

NIL, otherwise.

NOTES: Equivalent to (GE B A)

4.2.4.6. LT

CALL: (lt A B)

VALUE: T, if argument A's numeric value is less than argument B's.

NIL, otherwise.

NOTES: Similar to LISP predicate LESSP

Equivalent to (GT B A)

4.2.4.7. GTO

CALL: (gtO A)

VALUE: T, if argument A's numeric value is positive.

NIL, otherwise

NOTES: Equivalent to (GT A O)

4.2.4.8. GEO

CALL: (geO A)

VALUE: T, if argument A is non-negative.

NIL, otherwise

NOTES: Equivalent to (GE A O)

4.2.4.9. EQO

CALL: (eqO A)

VALUE: T, if argument A has value zero

NIL, otherwise

NOTES: Similar to LISP predicate ZEROP.

4.2.4.10. NEO

CALL: (neO A)

VALUE: T, if argument A's numeric value is non-zero.

NIL, otherwise

NOTES: Equivalent to (NE A O)

4.2.4.11. LEO

CALL: (leO A)

VALUE: T, if argument A's numeric value is negative or zero.

NIL, otherwise.

NOTES: Equivalent to (LE A O)

4.2.4.12. LTO

CALL: (ltO A)

VALUE: T, if argument A's numeric value is negative.

NIL, otherwise.

NOTES: Similar to LISP predicate MINUSP

Equivalent to (LT A 0) top

4.2.5. Floating point arithmetic functions

4.2.5.1. ADD

CALL: (add A1 ... An)

(add A) returns the value of A

(add) returns an integer zero

VALUE: The double precision floating point sum of the arguments A1 ... An. The imaginary parts of complex arguments are ignored.

NOTES: ADD never returns a complex result.

ADD accesses only the real parts of complex valued arguments.

4.2.5.2. SUB

CALLS: (sub A1 A2 ... An)

(sub A) returns the value of A

(sub) returns an integer zero

VALUE: The double precision floating point difference of the value of A1 and the sum of the argument values A2 ... An.

NOTES: SUB accesses only the real parts of complex valued arguments.

4.2.5.3. MULT

CALLS: (mult A1 ... An)

(mult A) returns the value of A.

(mult) returns an integer zero.

VALUE: The double precision floating point product of the argument values A1 ... An.

NOTES: MULT accesses only the real parts of complex valued arguments.

4.2.5.4. DIV

CALLS: (div A1 A2 ... An)

(div A) returns the value of A.

(div) returns an integer zero.

VALUE: The double precision floating point quotient of the value of argument A1 divided by the product of the argument values A2 ... An.

NOTES: DIV accesses only the real parts of complex valued arguments.

4.2.5.5. MIN

CALLS: (min A1 ... An)

(min A) returns the value of A.

(min) returns an integer zero.

VALUE: The double precision floating point minimum of the arguments values A1 ... An.

NOTES: MIN accesses only the real parts of complex valued arguments.

4.2.5.6. MAX

CALLS: (max A1 ... An)

(max A) returns the value of A.

(max) returns an integer zero.

VALUE: The double precision floating point maximum of the argument values A1 ... An.

NOTES: MAX accesses only the real parts of complex valued arguments.

4.2.6. Integer arithmetic and logical functions

4.2.6.1. IADD

CALLS: (iadd A1 ... An)

(iadd) returns an integer zero.

VALUE: The integer sum of argument values A1 ... An.

NOTES: A1 ... AN must all be integer valued.

4.2.6.2. ISUB

CALLS: (isub A1 A2 ... An)

(isub A) returns the value of A.

(isub) returns an integer zero.

VALUE: The integer difference of the value of A1 and the sum of the argument values A2 ... An.

NOTES: A1 ... AN must all be integer valued.

4.2.6.3. LAND

CALLS: (land A1 ... An)

(land A) returns the value of A.

(land) returns an integer zero.

VALUE: The logical AND of the argument values A1 ... An.

NOTES: A1 ... AN must all be integer valued.

4.2.6.4. LEQV

CALLS: (leqv A1 ... An)

(leqv A) returns the value of A.

(leqv) returns an integer zero.

VALUE: The logical EQUIVALENCE of the argument values A1 ... An.

NOTES: A1 ... AN must all be integer valued.

4.2.6.5. LOR

CALLS: (lor A1 ... An)

(lor A) returns the value of A.

(lor) returns an integer zero.

VALUE: The logical OR of the argument values A1 ... An.

NOTES: A1 ... AN must all be integer valued.

4.2.6.6. LXOR

CALLS: (lxor A1 ... An)

(lxor A) returns the value A.

(lxor) returns an integer zero.

VALUE: The logical EXCLUSIVE-OR of the argument values A1 ... An.

4.2.6.7. ASHIFT

CALL: (ashift N A)

VALUE: The (integer) value of argument A arithmetically shifted N places to the left.

NOTES: N and A are automatically converted to double precision integers. When N is negative ASHIFT shifts A arithmetically to the right.

-33 < N < 32.

4.2.6.8. LSHIFT

CALL: (lshift N A)

VALUE: The (integer) value of argument A logically shifted N places to the left.

NOTES: N and A are automatically converted to double precision integers. When N is negative LSHIFT shifts A logically to the right.

-33 < N < 32.

4.2.6.9. FLD

CALL: (fld SBIT LNG A)

VALUE: The LNG bits beginning with bit number SBIT of the (double precision integer) value of Argument A. The result is right justified.

NOTES: SBIT and LNG are automatically converted to single precision integers. A is automatically converted to a double precision integer.

-1 < SBIT < 33.

0 < LNG < 33.

0 < SBIT+LNG < 33.

4.2.6.10. OFLD

CALL: (ofld SBIT LNG A B)

VALUE: The integer value of argument A modified by replacing the LNG bits starting at bit SBIT of A with the right-most LNG bits of the integer value of argument B.

NOTES: SBIT and LNG are automatically converted to single precision integers.

A and B are atuomatically converted to double precision

integers.

-1 < SBIT < 33.

0 < LNG < 33.

0 < SBIT+LNG < 33.

4.2.7. Unary arithmetic functions

4.2.7.1. ABS

CALL: (abs A)

VALUE: The absolute value of the argument A.

NOTES: The value returned has the same type as the input argument A (unless A is complex)

ABS does not return the correct result for a complex argument because there is no square root routine. For a complex value A+Bi ABS returns A*A+B*B.

4.2.7.2. INC

CALL: (inc A)

VALUE: The value of argument A plus 1.

NOTES: The value returned is the same type as the input argument A. For a complex value A+Bi INC returns (A+1)+Bi.

4.2.7.3. DEC

CALL: (dec A)

VALUE: The value of argument A minus 1.

NOTES: The value returned is the same type as the input argument A. For a complex value A+Bi DEC returns (A-i)+Bi.

4.2.7.4. NEG

CALL: (neg A)

VALUE: The value 0 minus the value of argument A.

NOTES: The value returned is the same type as the input argument A. For a complex value A+Bi NEGATIVE returns (-A)+(-B)i.

4.2.8. Conversion functions

4.2.8.1. INT

CALLS: (int A)

(int) returns an integer zero.

VALUE: The truncated integer value of argument A.

If A is an integer value, that value is returned.

If A is a floating point value, the nearest integer value between A and zero is returned.

If A is a complex value, the nearest integer value between the real part of A and zero is returned.

4.2.8.2. REAL

CALLS: (real A)

(real) returns a double precision floating point zero.

VALUE: The double precision floating point value of argument A.

If A is an integer, its floating point equivalent is returned.

If A is a real value, that value is returned.

If A is a complex value, the real part of A is returned.

4.2.8.3. IMAGINARY

CALLS: (imaginary A)

(imaginary) returns a double precision floating zero.

VALUE: The imaginary part of the argument A.

If A is a complex value, the imaginary part of A is returned.

If A is an integer or real value, a floating point aero is returned.

4.2.8.4. COMPLEX

CALLS: (complex A B)

(complex A) returns A+0i

(complex) returns 0+0i

VALUE: The double precision complex value (REAL A)+(REAL B)i

4.2.9. Miscellaneous functions

4.2.9.1. UPTO

CALL: (upto L FCT)

VALUE: A list of the non-NIL values that result from applying FCT, a function of one argument, to successive members of the list L.

NOTES: If FCT is not a function, then LISP will request the user to supply a function.

UPTO is similar to the LISP function INTO, but UPTO filters out NILs.

Before a function that uses UPTO is compiled, the file "/lsp/fix" should be loaded. This allows the LISP compiler to compile UPTO calls of internal lambda

functions without declaring the free variables to be fluid.

UPTO can be loaded from the file "/lsp/upto".

4.2.9.2. CONCAT

CALL: (concat S1 ... Sn)

VALUE: The string that is the concatenation of the print names of S1 ... Sn.

NOTES: Si may be any LISP object that EXPLODE can handle. Si is usually a string, an atom, or a number.

CONCAT can be loaded from the file "/lsp/concat".

4.2.9.3. SQRT

CALL: (sqrt X)

VALUE: The double-precision square root of X.

EFFECT: SQRT uses Newton's method to calculate the square root of X. When X is negative, SQRT generates a "floating exception" and evaluates an (ERROR -10).

NOTES: SQRT can be loaded from the file "/lsp/sqrt" or "/lisp/sqrt".

4.2.9.4. EXP

CALL: (exp X)

VALUE: e**X

EFFECT: The double-precision quantity e**X is calculated and returned.

NOTES: EXP returns 0.0 when X is less than -88.

EXP takes a "floating exception" when X is greater than 88.

EXP can be loaded from the file "/lsp/exp" or

"/lisp/exp".

4.2.9.5. LOG

CALL: (log X)

VALUE: The double-precision natural (to the base e) logarithm of X.

NOTES: LOG can be loaded from the file "/lsp/log" or "/lisp/log".

4.2.9.6. LOG2

CALL: (log2 X)

VALUE: The double-precision logarithm to the base 2 of X.

NOTES: LOG2 can be loaded from the file "/lsp/log".

4.2.9.7. COS

CALL: (cos X)

VALUE: The cosine of X.

NOTES: COS can be loaded from the file "/lsp/sin" or "/lisp/sin".

4.2.9.8. SIN

CALL: (sin X)

VALUE: The sine of X.

NOTES: SIN can be loaded from the file "/lsp/sin" or "/lisp/sin".

4.2.9.9. ATAN

CALL: (atan X)

VALUE: The double-precision arctangent of X.

NOTES: The range of ATAN is -PI/2 to PI/2.

ATAN can be loaded from the file "/lsp/atan" or "/lisp/atan".

4.2.9.10. ATAN2

CALL: (atan2 X Y)

VALUE: The double-precision arctangent of X/Y.

NOTES: The range of ATAN2 is -PI to PI.

ATAN2 can be loaded from the file "/lsp/atan" or "/lisp/atan".

5. References

1. K. C. Hayes, XAP Users' Manual, Computer Science Technical Report TR-348, University of Maryland, January 1975.

2. R. L. Kirby, ULISP for PDP-11s with Memory Management, Computer Science Technical Report TR-546, University of Maryland, June 1977.

6. PDP 11 Function Index

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* PDP-11 Image Processing Software		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER TR-612 ✓
7. AUTHOR*(s)* Kenneth C. Hayes, Jr., Martin Herman, and Russell Smith		8. CONTRACT OR GRANT NUMBER*(s)* DAAG53-76C-0138 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December 1977
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

PDP-11/45
UNIX operating system
FORTRAN
LISP
Image processing

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The implementation of two different image handling systems for use on PDP 11/45 minicomputers is documented. Micro-XAP is an image access facility for UNIX FORTRAN users. Virtual Arrays are a more general image access facility for UNIX LISP users. This report serves as a user's guide for users who will interact with Micro-XAP or Virtual Arrays.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*